Quantum Computing and Shor's Algorithm

Matthew Hayward

February 17, 2005

Contents

1 Preface

This paper is intended to be a beginners introduction to the field of quantum computing. It is a very new field, and as a result there seems to be little good information for the true beginner. A short while ago I knew virtually nothing about quantum computing, as part of a project at the University of Illinois under the supervision of Dr. Roy Cambpell I was free to pursue the topic. I have attempted to put down what I learned in a concise and understandable manner. This paper will hopefully serve as an introduction to the rudiments of quantum computing and the specifics of Shor's algorithm for factoring large numbers.

To get the most out of this paper you should be interested in computer science, or physics, or both. Additionally you should be familiar with the following topics:

- Binary representation of numbers
- Complex numbers
- Vector mathematics

That being said there are appendices covering the bare essentials of those topics.

One of the most perplexing parts of quantum computing for the unfamiliar can be the nomenclature. If you find yourself unsure of a term, check the glossary, it may be in there.

2 Introduction

During the past forty years astounding advances have been made in the manufacture of computers. The number of atoms needed to represent a bit in memory has been decreasing exponentially since 1950. Likewise the number of transistors per chip, clock speed, and energy dissipated per logical operation have all followed their own improving exponential trends. This rate of improvement cannot be sustained much longer, at the current rate in the year 2020 one bit of information will requite only one atom to represent it. The problem is that at that size the behavior of a computer's components will be dominated by the principles of quantum physics. (Williams, Clearwater)

As components shrink to where their behavior will soon be dominated more by quantum physics than classical physics, researchers have begun to investigate the potential of these quantum behaviors for computation. These physical limitations of the classical computer, and the possibility that the quantum computer can perform certain useful tasks more rapidly than any classical computer drive the study of quantum computing.

3 The Classical Computer

3.1 Turing Machines

To understand how a quantum computer can be more "powerful" than a classical computer we should first define what we mean by powerful, and by classical computer. One of the people most directly responsible for the current concept of computing machines is Alan Turing. Others who contributed to the early development of computer science include Kurt Godel, Emil Post, and Alonso Church.

Turing and others proposed mathematical models for computing which allowed for the study of algorithms and in absence of any particular computer hardware. This abstraction has proved invaluable in the field of computer science. Turing's model is called a Turing machine. The design of the Turing machine is the following: The machine consists of an infinite tape divided into cells, each cell can contain a 1, 0, or blank. Additionally a head moves about the tape, reads and writes to the cells, may halt computation, and maintains an internal state which determines what action the head will take next. (Steane)

3.2 The Church-Turing Thesis

Turing proposed the following hypothesis:

> Every 'function which would naturally be regarded as computable' can be computed by the universal Turing machine.

it should be noted that there is ambiguity as to what, precisely, a function which would naturally be regarded as computable means. Due to this ambiguity, this statement is not subject to rigorous proof.

There is strong evidence for this hypothesis; many diverse models of computation have been shown to compute the same set of functions as a Turing machine, as yet there have been no counterexamples to the thesis.

This thesis gives us insight into the "power" of computing machines. If a computing device can solve all the problems a Turing machine can solve, then it is as powerful as a Turing machine.

3.3 Complexity Classes

Determining if a problem is computable at all is interesting, but this distinction is not fine enough to determine if a problem can realistically be solved by a physical computer in a reasonable amount of time. If a computation will take billions of years to compute, it may as well be uncomputable from the perspective of the pragmatist. An algorithm can be characterized by the number of operations and amount of memory it requires to compute an answer given an input of size N. These characterizations of the algorithm determine what is called the algorithms complexity. Specifically, the complexity of an algorithm is determined by looking at how the number of operations and memory usage

required to complete the program scales with the size of the input of the program. Computer Scientists have grouped problems into the following complexity classes:

P: Polynomial time, the running time of the given algorithm is in the worst case some polynomial function of the size of the input. For example if an algorithm with an input size of 10 bits which took $10^4 + 7 * 10^2 + 1001$ operations to compute is a polynomial time algorithm.

NP: Nondeterministic polynomial time, a candidate for an answer can be verified as a correct answer or not in polynomial time.

NP-complete: A set of problems such that if any member is in P, the the set P equals the set NP. (Cormen, Leiserson, Rivest)

Problems which can be solved in polynomial time or less are generally deemed to be "tractable." Problems which require more than polynomial time are usually considered to be "intractable," for example an algorithm which would take 2^n operations for an input size of n would be considered intractable, as the number of operations grows exponentially with the input size, not polynomially.

It should be noted at this point that in Shor's Algorithm, we will be given n, a number to be factored. In n's most compact representation, it will take $\lceil \log n \rceil$ bits to be encoded. When discussing the running time of algorithms, we must always speak in reference to the encoded input size. For Shor's algorithm (or any algorithm whose input is an integer), the "input size" grows logarithmically with the value of n to be factored.

As an aside, whether or not P and NP describe the same set of problems is one of the most important open questions in computer science. Should a polynomial time algorithm for any NP-complete problem be discovered, then we would know that P and NP are the same set. Many NP-complete problems have known algorithms which can compute their solutions in exponential time, whether there exists any polynomial time algorithms is unknown.

When discussing the complexity of a problem it is important to distinguish between there being no *known* algorithm to compute it in polynomial time, and there being *no* algorithm to compute it in polynomial time. There are many problems whose best *known* algorithm requires an exponential number of steps to compute a solution. These problems are generally considered to be intractable. Determining the prime factors of a large number is one such problem, and its assumed intractability is the bases for most cryptography.

The interest in quantum computing is twofold. First it is of interest if a quantum computer can solve problems which are uncomputable on a classical computer, and second it is of interest if a quantum computer can make problems thought of as intractable, tractable.

4 The Quantum Computer

4.1 Quantum Physics

When considering the possible power of a computer which makes use of the results of quantum physics, it is helpful to know a little of quantum physics itself. Quantum physics arose from the failure of classical physics to offer correct predictions on the behavior of photons and other elementary particles. Quantum physics has since been under intense scrutiny, as some of its predictions seem very strange indeed. Nevertheless experiments verify the same strange behavior which leads skeptics to challenge the veracity of Quantum physics.

4.2 The Classical Bit

To understand the ways in which a quantum computer is different from a classical computer you must first understand the rudiments of the classical computer. The most fundamental building block of a classical computer is the bit. A bit is capable of storing one piece of information, it can have a value of either 0 or 1. Any amount of information can be encoded into a list of bits. In a classical computer a bit is typically stored in a silicone chip, or on a metal hard drive platter, or on a magnetic tape. As this paper is written about 10^{10} atoms are typically used to store one bit of information. The smallest conceivable storage for a bit involves a single elementary particle of some sort. For example, any particle with a spin-1/2 characteristic can be characterized by its spin value, which when measured is either $+1/2$ or $-1/2$. We can thus encode 1 to be $+1/2$ and 0 to be $-1/2$, and if we assume we can measure and manipulate the spin of such a particle then we could theoretically use this particle to store one bit of information. If we were to try to use this spin-1/2 particle as a classical bit, one that is always in the 0 or 1 state, we would fail. We would be trying to apply classical physics on a scale where it simply is not applicable. This single spin-1/2 particle will instead act in a quantum manner. (Williams, Clearwater)

This spin-1/2 particle which behaves in a quantum manner could be the fundamental building block of a Quantum computer. We could call it a qubit, to denote that it is analogous in some ways to a bit in a classical computer. Just as a memory register in a classical computer is an array of bits, a quantum memory register is composed of several qubits. There is no particular need for the spin-1/2 particle, equally well we could use a Hydrogen atom, and designate its electron being measured in the ground state to be the 0 state, and it being in the first excited state to be the 1 state. For simplicity I will discus only qubits from here on, ignoring their particular implementation.

4.3 State Vectors and Dirac Notation

We wish to know exactly how the behavior of our qubit differs from a that of a classical bit. Recall that a classical bit can store either a 1 or a 0, and when measured the value observed will always be the value stored. Quantum physics

states that when we measure the qubit we will determine that it is in the 1 or the 0 state. In this manner our qubit is not different from a classical bit. The differences between the qubit and the bit come from what sort of information a qubit can store when it is not being measured.

According to quantum physics we may describe that state of our qubit by a state vector in a Hilbert Space. In general the mathematical term space refers to a something which depends on many independent coordinates which can be defined by a set of perpendicular axes, one for each independent variable. For example you are probably familiar with the x, y, z coordinate system where the x, y, and z axes are mutually perpendicular real number lines, which coincide at the point $x = 0$, $y = 0$, $z = 0$.

A Hilbert Space is a special kind of space, it has the properties that it is a complex vector space, and it is a linear vector space. A complex vector space is one where the lengths of the vectors within the space are described by complex numbers. Complex numbers are numbers which take the form $a + i * b$, where a and b are real numbers, and i is the square root of negative one. A linear vector space is one where you may add and multiply vectors that lie within the space and the resulting vector will still lie within the space. (Williams, Clearwater)

In the Hilbert Space for a quantum system's state vector, we choose to define these perpendicular axes to correspond to each possible state that the system can be measured in. Our Hilbert Space for a single qubit will have two perpendicular axes, one corresponding to the qubit being in the 1 state, and the other corresponding the qubit being in the0 state. These states which the vector can be measured to be are referred to as "eigenstates." The vector which exists somewhere in this space which represents the state of our qubit is called the "state vector." The projection of the state vector onto one of the axes shows the contribution of that axis' eigenstate to the whole state.

In general, the state of a qubit can be any combination of the base states. In this manner a qubit it totally unlike a bit, for a bit can exist in only the 0 or 1 state, but the qubit can exist, in principle, in any combination of the 0 and 1 state, and is only constrained to be in the 0 or 1 state upon measurement.

Now I will introduce some standard notation for state vectors in Quantum physics. The state vector is written with the following way and called a "ket vector" $|\psi\rangle$. Where ψ is a list of numbers which contain information about the projection of the state vector onto its base states. The term ket and this notation come from the physicist Paul Dirac who wanted a concise shorthand way of writing formulas that occur in Quantum physics. These formulas frequently took the form of the product of a row vector with a column vector. Thus he referred to row vectors as "bra vectors" represented as $\langle y|$. The product of a "bra" and a "ket" vector would be written $\langle y|x\rangle$, and would be referred to as a "bracket." (Williams, Clearwater)

There is nothing special about this vector notation, you may think of any state vector being written as a letter with a line over it, or as a bold letter, as vectors are normally denoted in mathematical literature. If you do further reading in this area you will assuredly come across the bra and ket notation, which is why it is presented.

4.4 Superposition and Eigenstates

Earlier I said that the projection of the state vector onto one of the perpendicular axes of its Hilbert Space shows the contribution of that axis' eigenstate to the whole state. You may wonder what is meant by the "whole state." You would think (and rightly so, according to classical physics) that our qubit could only exist entirely in one of the 1 or 0 states, and accordingly that its state vector could only exist lying completely along one of its coordinate axes. It would seem that if the particle's axes are called x and y, and the state vector's x coordinate, which denotes the contribution of the 0 state, and y coordinate which denotes the contribution of the 1 state, the only valid states should be; (1,0), or (0,1).

That seems perfectly reasonable, but it simply is not correct. According to quantum physics a quantum system can exist in a mix of all of its allowed states simultaneously. This mixing of states is called quantum superposition, and it is key to the power of the quantum computer. While the physics of superposition is not simple at all, mathematically it is not difficult to characterize.

4.5 The Qubit

Let x_1 be the eigenstate corresponding to the 1 state, and let x_0 be the eigenstate corresponding to the 0 state. Let X be the total state of our state vector, and let w_1 and w_0 be the complex numbers that weight the contribution of the base states to our total state, then in general:

$$|X\rangle = w_0 * |x_0\rangle + w_1 * |x_1\rangle \equiv (w_0, w_1)$$

At this point it should be remembered that w_0 and w_1, the weighting factors of the base states are complex numbers, and that when the state of X is measured, we are guaranteed to find it to be in either the state:

$$0 * |x_0\rangle + w_1 * |x_1\rangle \equiv (0, w_1)$$

or the state

$$w_0 * |x_0\rangle + 0 * |x_1\rangle \equiv (w_0, 0)$$

This is analogous to a system you should be more familiar with, a vector with real weighting factors in the a two dimensional plane. Let the base states for this two dimensional plane be the unit vectors x, and y. In this case we know that the state of any vector V can be described in the following manner:

$$V = x_0 * x + y_0 * y \equiv (x_0, y_0)$$

We can further restrict our state vector to be a unit vector in a Hilbert space. It is not necessary from a physics perspective for the state vector to be a unit vector (by which I mean it has a length of 1), but it makes for easier calculations further on, so I will assume from here on out that the state vector has length 1. This assumption does not invalidate any claims about the behavior of the state vector. To see how to convert a state vector of any length to length 1 see appendix A.

4.6 The Quantum Memory Register

We have considered a two state quantum system, a qubit. However a quantum system is by no means constrained to be a two state system. Much of the above discussion for a 2 state quantum system is applicable to a general n state quantum system.

In an n state system our Hilbert Space has n perpendicular axes, or eigenstates, which represent the possible states the system can be measured in. As with the two state system, when we measure our n state quantum system, we will always find it to be in exactly one of the n states, and not a superposition of the n states. The system is still allowed to exist in any superposition of the n states while it is not being measured.

Mathematically if two state quantum system with coordinate axes x_0, x_1 can be fully described by:

$$|X\rangle = w_0 * |x_0\rangle + w_1 * |x_1\rangle \equiv (w_0, w_1)$$

Then an n state quantum system with coordinate axes $x_0, x_1, \ldots, x_{n-1}$ can be fully described by:

$$|X\rangle = \sum_{k=0}^{n-1} w_k * |x_k\rangle$$

In general a quantum system with n base states can be represented by the n complex numbers w_0 to w_{n-1}. When this is done the state may be written as:

$$|X\rangle = \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_{n-1} \end{pmatrix}$$

Where it is understood that w_k refers to the complex weighting factor for the k'th eigenstate.

Using this information we can construct a quantum memory register out of the qubits described in the previous section. We may store any number x in our quantum memory register as long as we have enough qubits, just as we may store any number x in a classical register as long as we have enough classical bits to represent that number. The state of the quantum register with n states is give by the formula above. Note that in general a quantum register composed of m qubits requires 2^m complex numbers to completely describe its state: an m qubit register can be measured to be in one of 2^m states, and each state requires one complex number to represent the projection of that total state onto that state. In contrast a classical register composed of m bits requires only m integers to fully describe its state.

This means that is some sense one can store an exponentially greater amount of information in a quantum register than in a classical memory register of the same number of (q)bits. Here we see some of the first hints that a quantum computer can be exponentially more powerful than a classical computer in some

respects. Recall that from our discussion of complexity that problems whose best known algorithms yield a solution in polynomial time are generally thought of as being tractable, and that problems whose best known algorithms take exponential time are thought of as intractable. If a quantum computer really is exponentially faster than a classical computer, then some intractable problems may become tractable! This is a large part of the motivation for the study of quantum computing.

4.7 Probability Interpretation

Now that we know how to represent our state vector as a superposition of states, but we know that we can only measure the state vector to be in one of the base states, we must determine what happens when we measure the state vector. We know from quantum physics that given an initial condition the state vector will evolve in time in accordance with Schrödenger's equation:

$$ih\frac{\partial |X(t)\rangle}{\partial t} = H(t)\,|X(t)\rangle$$

Where i is the square root of negative one, h is $1.0545 * 10^{-34}$ Js, and H is the Hamiltonian operator, which is determined by the physical characteristics of the system being evolved.

In Dirac notation this expression is:

$$ih\frac{\partial w_i(t)}{\partial t} = \sum_j H(t)_{ij} * w_j(t)$$

The evolution of the state vector through time would appear to be continuous, but these equations only apply to a quantum mechanical system evolving in isolation from the environment. The only way to observe the state of the state vector is to in some way cause the quantum mechanical system to interact with the environment. When the state vector is observed it makes a sudden discontinuous jump to one of the eigenstates. When the quantum mechanical system interacts with the outside environment and measured the state vector is said to have collapsed. (Williams, Clearwater)

To perform any useful calculations we must be able so say something about which base state a quantum mechanical system will collapse into. The probability that the state vector will collapse into the j'th eigenstate is given by $|w_j|^2$ which is defined to be $a_j^2 + b_j^2$ when $w_j = a_j + i * b_j$ and w_j is the complex projection of the state vector onto the j'th eigenstate. In general the chance of choosing any given state is

$$Prob(j) = \frac{|w_j|^2}{\sum_{k=0}^{n-1} |w_k|^2}$$

but as mentioned earlier we will insist on having the state vector of length one, and in this case the probability expression simplifies to $Prob(j) = |w_j|^2$.

We now know how to describe an n state quantum system, which can be placed in an arbitrary superposition of states. We also know how measuring this superposition yield a given base state with a given probability. This is all that we need to understand about our quantum memory register to be able to simulate its behavior.

4.8 Quantum Parallelism

The study of quantum computing is relatively new, most give credit to Richard Feynman for being the first to suggest that there were tasks that a quantum computer could perform exponentially faster than a classical computer. Feynman showed that a Turing machine could not simulate certain quantum mechanical systems without performing an exponential number of steps in relation to the number of elements of the system. At the same time he hinted that perhaps by using a device whose behavior was inherently quantum in nature one could simulate such a system without this exponential slowdown. (Feynman)

Most modern quantum algorithms rely on something called quantum parallelism. Quantum parallelism arises from the ability of a quantum memory register to exist in a superposition of base states. A quantum memory register can exist in a superposition of states, each component of this superposition may be thought of as a single argument to a function. A function performed on the register in a superposition of states is thus performed on each of the components of the superposition. Since the number of possible states is 2^n where n is the number of qubits in the quantum register, you can perform in one operation on a quantum computer what would take an exponential number of operations on a classical computer. This is fantastic, but the more superposed base states that exist in you register, the smaller the probability that you will measure the result of your function on any particular base state becomes.

Suppose that you are using a quantum computer to calculate the function $\mathcal{F}(x) = 2{*}x \bmod 7$, where x takes on the integers between 0 and 7 inclusive. You could prepare a quantum register that was in a equally weighted superposition of the states 0-7. Then you could perform the $2 * x \bmod 7$ operation once, and the register would contain the equally weighted superposition of $0, 2, 4, 6, 1, 3, 5, 0$, these being the outputs of the function $2 * x \bmod 7$ for inputs 0 - 7. When measuring the quantum register you would have a 2/8 chance of measuring 0, and a 1/8 chance of measuring any of the other outputs. It would seem that this sort of parallelism is not useful, as the more we benefit from parallelism the less likely we are to measure a value of a function for a particular input. Some clever algorithms have been devised, most notably by Peter Shor and L. K. Grover which succeed in using quantum parallelism on a function where they are interested in some property of all the inputs, not just a particular one.

4.9 The Power of the Quantum Computer

Given the possible power of quantum parallelism, much work has been done to show formally with mathematical proofs how quantum computers differ from

classical ones in their power to compute things. Here is a short list of some of the landmarks in the study of the power of quantum computers.

In 1980 Paul Benioff offered a classical Turing machine which used quantum mechanics in its workings, thus showing that theoretically a quantum computer was at least as powerful as a classical computer. (Benioff)

David Deutsch and Richard Jozsa showed in a paper in 1992 that there was an algorithm that could be run in poly-log time on a quantum computer, but required linear time on a deterministic Turing machine. This may have been the first example of a quantum computer being shown to be exponentially faster than a deterministic Turing machine. Unfortunately for the quantum computer, the problem could also be solved in poly-log time in a probabilistic Turing machine, a Turing machine which is capable of making a random choice. (Deutsch, Jozsa)

Also in 1992 Andre Berthiaume proved that $P \subset QP$, where P is a complexity class as mentioned earlier and QP corresponds to problems which can be solved in worst case polynomial time by a quantum computer, so with regards to tractability a quantum computer is in some sense more powerful than any classical computer. (Berthiaume, Brassard)

5 Quantum Algorithms

5.1 Introduction to Shor's Algorithm

By the early nineties it was know that a quantum computer could be faster than any classical computer for certain problems. Nonetheless these observations were largely driven by academic curiosity. There was not much motive for people to spend lots of money or time trying to build a quantum computer.

This changed in 1994 when Peter Shor, a scientist working for Bell Labs, devised a polynomial time algorithm for factoring large numbers on a quantum computer. This discovery drew great attention to the field of quantum computing.

5.2 Motivation for Shor's Algorithm

The algorithm was viewed as important because the difficulty of factoring large numbers is relied upon for most cryptography systems. If an efficient method of factoring large numbers is implemented most of the current encryption schemes would be worthless. While it has not been proven that factoring large numbers can not be achieved on a classical computer in polynomial time, the fastest algorithm publicly available for factoring a large number n (whose representation has $\lceil \log n \rceil$ bits) runs in $O(e^{c(\log n)^{1/3} * (\log \log n)^{2/3}})$, or exponential time. In contrast Shor's algorithm runs in $O((\log n)^2 * \log \log n)$ on a quantum computer, and then must perform $O(\log n)$ steps of post processing on a classical computer. Overall then this time is polynomial. This discovery propelled the study of quantum computing forward, as such an algorithm is much sought after. (Shor)

5.3 Overview of Shor's Algorithm

Shor's algorithm hinges on a result from number theory: the function $\mathcal{F}(a) = x^a \bmod n$ is a periodic function when x is an integer coprime to n. In the context of Shor's algorithm n will be the number we wish to factor. When two numbers are coprime it means that their greatest common divisor is 1.

Calculating this function for an exponential number of a's would take exponential time on a classical computer. Shor's algorithm utilizes quantum parallelism to perform the exponential number of operations in one step.

Since $\mathcal{F}(a)$ is a periodic function, it has some period r. We know that $x^0 \bmod n = 1$ (since $x^0 = 1$), and therefore $x^r \bmod n = 1$, and $x^{2r} \bmod n = 1$, and so on.

Given this information and through the following algebraic manipulation:

$$x^r \equiv 1 \bmod n$$

$$(x^{r/2})^2 = x^r \equiv 1 \bmod n$$

$$(x^{r/2})^2 - 1 \equiv 0 \bmod n$$

and if r is an even number

$$(x^{r/2} - 1)(x^{r/2} + 1) \equiv 0 \bmod n$$

We can see that the product $(x^{r/2} - 1)(x^{r/2} + 1)$ is an integer multiple of n, the number to be factored. So long as $\left|x^{r/2}\right|$ is not 1, then at least one of $(x^{r/2} - 1)$, $(x^{r/2} + 1)$ must have a nontrivial factor in common with n. So by computing $\gcd(x^{r/2} - 1, n)$, and $\gcd(x^{r/2} + 1, n)$, we will obtain a factor of n, where gcd is the greatest common denominator function.

Shor's algorithm tries to find r, the period of $x^a \bmod n$, where n is the number to be factored by Shor's algorithm, and x is an integer coprime to n. To do this Shor's algorithm creates a quantum memory register with two parts. In the first part the algorithm places a superposition of the integers which are to be a's in the $x^a \bmod n$ function. We will choose our a's to be the integers 0 through $q - 1$, where q is the power of two such that $n^2 \leq q < 2n^2$. Then the algorithm calculates $x^a \bmod n$, where a is the superposition of the states, and places the result in the second part of the quantum memory register.

Remember, the number n is represented by a $\lceil \log n \rceil$ bit string. We must calculate $x^a \bmod n$ an exponential number of times, with respect to the length of the encoded input to the algorithm (although it is a polynomial number of times with respect to the value of n).

Next the algorithm measures the state of the second register, the one that contains the superposition of all possible outcomes for $x^a \bmod n$. Measuring this register has the effect of collapsing the state into some observed value, say k. It also has the side effect of projecting the first part of the quantum register into a state consistent with the value measured in the second part. So: measurement of the second part results in exactly one value, and causes the other partition

to collapse into a superposition of the base states consistent with the value observed in the second part.

After this measurement the second part of the register contains the value k, and the first part of the register contains a superposition of the base states whose evaluation in $x^a \bmod n$ produce k. We know $x^a \bmod n$ is a periodic function, therefore the first part of the register will contain the values c, $c + r$, $c + 2r \ldots$, where c is the lowest integer such that $x^c \bmod n = k$.

The next step is to perform a discrete Fourier transform on the contents of first part of the register. The application of the discrete Fourier transformation has the effect of peaking the probability amplitudes of the first part of the register at integer multiples of the quantity q/r.

Measuring the first part of the quantum register will yield an integer multiple of the inverse of the period with high probability. Once this number is retrieved from the quantum memory register, a classical computer can do analysis of this number, make a guess as to the actual value of r, and from that compute the possible factors of n. This post processing is covered in more detail below. (Shor)

5.4 Steps to Shor's Algorithm

Shor's algorithm for factoring a given integer n can be broken into some simple steps.

1. Determine if the number n is a prime, a even number, or an integer power of a prime number. If it is we will not use Shor's algorithm. There are efficient classical methods for determining if a integer n belongs to one of the above groups, and providing factors for it if it does. This step would be performed on a classical computer.

2. Pick a integer q that is the power of 2 such that $n^2 \leq q < 2n^2$. This step would be done on a classical computer.

3. Pick a random integer x that is coprime to n. When two numbers are coprime it means that their greatest common divisor is 1. There are efficient classical methods for picking such an x. This step would be done on a classical computer.

4. Create a quantum register and partition it into two sets, register one and register two. Thus the state of our quantum computer can be given by: $left|\text{reg1, reg2}\rangle$. Register one must have enough qubits to represent integers as large as $q - 1$. Register two must have enough qubits to represent integers as large as $n - 1$.

5. Load register one with an equally weighted superposition of all integers from 0 to $q - 1$. Load register two with the 0 state. This operation would be performed by our quantum computer. The total state of the quantum

memory register at this point is:

$$\frac{1}{\sqrt{q}} \sum_{a=0}^{q-1} |a, 0\rangle$$

6. Apply the transformation $x^a \bmod n$ to for each number stored in register one and store the result in register two. Due to quantum parallelism this will take only one step, as the quantum computer will only calculate $x^{|a\rangle} \bmod n$, where $|a\rangle$ is the superposition of states created in step 5. This step is performed on the quantum computer. The state of the quantum memory register at this point is:

$$\frac{1}{\sqrt{q}} \sum_{a=0}^{q-1} |a, x^a \bmod n\rangle$$

7. Measure the second register, and observe some value k. This has the side effect of collapsing register one into a equal superposition of each value a between 0 and $q-1$ such that

$$x^a \bmod n = k$$

This operation is performed by the quantum computer. The state of the quantum memory register after this step is:

$$\frac{1}{\sqrt{||A||}} \sum_{a'=a' \in A} |a', k\rangle$$

Where A is the set of a's such that $x^a \bmod n = k$, and $||A||$ is the number of elements in that set.

8. Compute the discrete Fourier transform on register one. The discrete Fourier transform when applied to a state $|a\rangle$ changes it in the following manner:

$$|a\rangle = \frac{1}{\sqrt{q}} \sum_{c=0}^{q-1} |c\rangle * e^{2\pi i a c/q}$$

This step is performed by the quantum computer in one step through quantum parallelism. After the discrete Fourier transform our register is in the state:

$$\frac{1}{\sqrt{||A||}} \sum_{a' \in A} \frac{1}{\sqrt{q}} \sum_{c=0}^{q-1} |c, k\rangle * e^{2\pi i a' c/q}$$

9. Measure the state of register one, call this value m, this integer m has a very high probability of being a multiple of q/r, where r is the desired period. This step is performed by the quantum computer.

10. Take the value m, and on a classical computer do some post processing which calculates r based on knowledge of m and q. There are many ways to do this post processing, they are complex are are omitted for clarity in presentation of the quantum core of Shor's Algorithm. This post processing is done on a classical computer.

11. Once you have attained r, a factor of n can be determined by taking $\gcd(x^{r/2}+1, n)$ and $\gcd(x^{r/2}-1, n)$. If you have found a factor of n, then stop, if not go to step 4. This final step is done on a classical computer.

Step 11 contains a provision for Shor's algorithm failing to produce the factors of n. Shor's algorithm can fail for multiple reasons, for example the discrete Fourier transform could be measured to be 0 in step 9, making the post processing in step 10 impossible. At other times the algorithm will sometimes find factors of 1 and n, which is correct but not useful. (Williams, Clearwater)

5.5 Other Quantum Algorithms

Shor's algorithm is not the only algorithm that seems to be better on a quantum computer than any classical computer for a problem which is considered to be useful. In 1994 L. K. Grover, also of Bell Labs, devised an algorithm to find an item in an unsorted list of N elements in $.758\sqrt{N}$ operations. No classical algorithm can guarantee finding the item in less than N operations, and in the average case it would take $N/2$ operations. (Grover)

6 A Simulation of Shor's Algorithm on a Classical Computer

In order to make the steps of Shor's algorithm more concrete I simulated the operation of a quantum computer performing Shor's algorithm. This simulation of a quantum system incurs an exponential number of operations based on the input size, thus the simulation is only tolerable to observe for relatively small numbers.

The simulation implements all of the steps in Shor's algorithm, and successfully factors numbers. All source code is written in C++ and can be found in appendix D.

6.1 Introduction to the Code for the Simulation

The code base for the simulation of Shor's algorithm consists of four files.

complex.cpp: This file contains the simple complex number class for storing state information about the quantum memory register in the simulation.

qureg.cpp: This file contains the quantum register class that simulates the behavior of the quantum memory register in Shor's algorithm. It is generic enough that it may be made to simulate any quantum memory register, although

what happens in the event of a collapse not due to the direct measurement of the register is left to the programmer to perform. An example of such a collapse that is the collapse of the first partition of Shor's quantum memory register in step 7 of his algorithm.

shor.cpp: This is the main body of the simulation, which contains the Steps of Shor's algorithm as described above.

util.cpp: This is a library of useful functions used by shor.cpp

6.2 The Complex Number Class

The probability amplitudes, or the lengths, of a vector in a Hilbert Space are in general complex. The state of our quantum memory register may be thought of as a vector in a Hilbert Space. Each complex number can be set to a complex value, which is stored internally as a real part, and an imaginary part, both of which are double precision floating point numbers.

6.3 The Quantum Memory Register Class

The qureg class, the quantum memory register, is defined in qureg.cpp. When a quantum memory register is created it takes as input the number of qubits it is composed of. A quantum memory register with n qubits requires 2^n complex numbers to represent it. This is because a register of n bits can exist in any one of 2^n base states, and a quantum register may exist in any superposition of those base states. I use one complex number per base state to describe the probability of that state being measured.

For example, if a quantum register has 3 bits it can be measured to exist in any one of the following states:

$$|0,0,0\rangle$$

$$|0,0,1\rangle$$

$$|0,1,0\rangle$$

$$|0,1,1\rangle$$

$$|1,0,0\rangle$$

$$|1,0,1\rangle$$

$$|1,1,0\rangle$$

$$|1,1,1\rangle$$

Taking these values as binary numbers with the most significant bit the left most bit, then these states correspond to the numbers 0,1,2,3,4,5,6,7. There are $2^3 = 8$ possible base states.

The probability of measuring the j'th state whose complex amplitude is w_j is $|w_j|^2 / \sum_j |w_j|^2$. I ensure throughout that algorithm that $\sum_j |w_j|^2$ is 1, so the probability of measuring the j'th state is simply $|w_j|^2$. The base states of our n

bit quantum register can be thought of as being the integers 0 through 2^{n-1}. In code the quantum register of size n will be represented by an array of complex numbers of size 2^n. The value stored at the j'th array position is the complex probability amplitude associated with measuring the number j. For example, a quantum memory register of size 2 that contains an equal superposition of the numbers 0-3 would be represented by:

$$State[0] = 1/2 + i * 0$$

$$State[1] = 1/2 + i * 0$$

$$State[2] = 1/2 + i * 0$$

$$State[3] = 1/2 + i * 0$$

Observe that the probability of measuring any given state is $|w_j|^2 = \frac{1}{2}^2 + 0^2 = 1/4$.

If we attempt to measure this register, we will with equal probability measure one of $0, 1, 2, 3$. Let us assume we measure the state and observe the value 2, this has the effect of collapsing all probabilities not measured to 0, so the new state of our quantum register is:

$$State[0] = 0 + i * 0$$

$$State[1] = 0 + i * 0$$

$$State[2] = 1 + i * 0$$

$$State[3] = 0 + i * 0$$

If we were to measure this register again we would attain 2 without fail.

Sometimes during the course of operation we cause a state to exist in the quantum memory register where the sum over the squares of the probability amplitudes is not 1, for example a quantum register could be in the state:

$$State[0] = 1 + i * 0$$

$$State[1] = 1 + i * 0$$

We would expect that when measured 0 and 1 would be equally likely outcomes, however we would also like for simplicity to have $|w_j|^2$ be the probability of measuring the j'th state. To accommodate this desire the qureg class contains a subroutine which will normalize the probability amplitudes, such that sum of the $|w_j|^2$ over all j's is one. The qureg class also allows the state of the register to be set to any arbitrary state.

The quantum memory register class has two member functions which do things that a real quantum memory register could not do. The first such function is the Dump() function which dumps the entire state information of the register without collapsing it, and is included for debugging. The second such function is the GetProb(state) which gets the probability amplitude of any given state.

The GetProb function is used in the main program to calculate the discrete Fourier transform in step 8. While a quantum computer could not use this information in step 8, a quantum computer would not need to. In a quantum computer the Fourier transformation would take place in one step, and the transform function would take as its argument a state which could in general be any superposition of base states. The GetProb function is used to simulate the application of a function which takes a superposition of states as its argument.

6.4 The Simulation of Shor's Algorithm

The implementation of Shor's algorithm found in shor.cpp follows the steps outlined in the description of Shor's algorithm found above. There are some significant differences in the behavior of the simulator and the behavior of a actual quantum computer.

The simulator uses 2^{n+1} double precision floating point numbers to represent the state of the quantum register. It uses one Complex object to represent the probability amplitude of each of the 2^n eigenstates of a n bit quantum memory register, and each Complex object uses two double precision floating point numbers. On a machine in which a double precision floating point number is represented with 64 bits the simulator uses approximately 2^{n+7} classical bits to represent the state of the quantum memory register, where n is the number of bits required to represent the number the simulator is trying to factor. A real quantum computer would use exactly n qubits as its memory register.

A second large difference is that during the modular exponentiation step of Shor's algorithm (Step 6 above) a quantum computer would perform one operation of $x^a \bmod n$, where a is the superposition of states in register 1. The simulation must calculate the superposition of values caused by calculating $x^a \bmod n$ for $a = 0$ through $q-1$ iteratively. The simulation also stores the result of each modular exponentiation, and uses that information to collapse register 1 in step 7 in Shor's algorithm. A quantum computer would not be capable of performing this bookkeeping, as examining any particular result would collapse the existing superposition to be placed in register 2 at the end of step 6 of Shor's algorithm. A quantum computer would have no need whatsoever to do such bookkeeping, as when when register 2 is measured in step 7, the collapse of register 1 is an automatic and unavoidable consequence of the measurement. A analogous argument follows for the use of the get probability function in step 8 of Shor's algorithm for calculating the discrete Fourier transform.

Aside from these differences, which are necessitated by the inability of a classical computer to accurately simulate the behavior of a quantum mechanical system, the operations are performed by the shor.cpp program are identical to those called for in the description of Shor's algorithm.

After we perform the steps of Shor's algorithm, with a final measurement of register 1 in step 9 we obtain some integer m, which has a high probability of being an integer multiple λ of q/r, where λ is some integer, and r is the period of $x^a \bmod n$ that we are trying to find, so that we may calculate $x^{r/2} - 1 \bmod n$ and $x^{r/2} + 1 \bmod n$ in an effort to find numbers which share factors with n.

In the post processing step shor.cpp takes the integer m and divides it by q, thus yielding some number c which is approximately equal to λ/r, where λ is an integer, and r is the desired period. Then using a helper function it calculates the best rational approximation to c which has a denominator that is less than or equal to q (recall that q is the power of two such that $n^2 \leq q < 2n^2$, where n is the number to be factored by Shor's algorithm).

We take this denominator to be our period. Shor's algorithm can only use even periods in determining factors of n, and so we check to see if r is even, if not we check to see if doubling the period would still yield a period less than q, if so we double our guessed period.

With the resultant guess for the period we calculate $x^{r/2} - 1 \bmod n$ and $x^{r/2} + 1 \bmod n$, calling these values a and b, we compute the greatest common denominator of a and n, and b and n, to see if we have attained a nontrivial factor of n.

6.5 Utility Functions for the Simulation

The util.cpp file contains functions which are called from shor.cpp. Many of the functions are written to avoid overflow when calculating functions in which there are intermediate steps which may have large values. For example while factoring 15, $x^{255} \bmod 15$ is calculated, but the modular exponentiation function in util.cpp calculates it in such a manner that x^{255} need not be explicitly calculated.

7 Conclusion

Between the Turing machine and the Church-Turing Thesis a strong foundation was made for study of that what was computable and uncomputable. Complexity analysis allows for a more granular distinction between the tractability of any particular problem on a Turing machine.

The components in classical computers are rapidly shrinking to a size where quantum behavior is inevitable. Through the principle of superposition in quantum systems we can create useful memory components that are on the scale of an atom or smaller. These quantum memory registers may facilitate exponential computational speed increases by taking advantage of quantum parallelism.

Peter Shor has provided an algorithm which makes factoring large numbers tractable, and in doing so has drawn great attention to the field of quantum computing. Due to Shor's algorithm, we may someday have to turn to other means of encrypting data than currently employed. L. K. Grover's database search algorithm shows another noteworthy task that a quantum computer can perform faster than any classical computer.(Brassard) The efforts to build a functional quantum memory register are in the most preliminary stages.

Operational quantum computers are by no means an inevitable consequence of this research. It may be that the problems surrounding keeping a quantum memory register isolated from any disturbance long enough for a calculation

to take place will be insurmountable. In any case, quantum computing will remain an exciting topic for experimentalists and theorists alike for years to come. Hopefully this paper, and the simulation of Shor's algorithm have been as enlightening and fun for the reader as they were for the author.

8 Bibliography

Benioff, p. "The Computer as a Physical System: A Microscopic Quantum Mechanical Hamiltonian Model of Computers as Represented by Turing Machines," *Journal of Statistical Physics*, Vol. 22 (1980), pp. 563-591.

Berthiaume, Andre and Brassard, Gilles. "The quantum Challenge to Complexity Theory," *Proceedings of the 7th IEEE Conference on Structure in Complexity Theory* (1992), pp. 132-137.

Brassard, Gilles. "Searching a Quantum Phone Book," *Science*, 31 January 1997.

Cormen, Thomas H., Leiserson, Charles E., and Rivest, Ronald L. "Introduction to Algorithms," St. Louis: McGraw-Hill, 1994.

Deutsch, David and Jozsa, Richard. "Rapid Solution of Problems by Quantum Computation," *Proceedings Royal Society London*, Vol. 439A (1992), pp. 553-558.

Feynman, Richard. "Simulating Physics with Computers," *Optics News* Vol. 11 (1982), pp. 467-488.

Grover, L. K. "A Fast Quantum Mechanical Algorithm for Database Search," *Proceedings of the 28'th Annual ACM Symposium on the Theory of Computing* (1996), pp. 212-219.

Shor, Peter. "Algorithms for Quantum Computation: Discrete Logarithms and Factoring," *Proceedings 35th Annual Symposium on Foundations of Computer Science* (1994), pp. 124-134.

Steane, Andrew. "Quantum Computing," *Reports on Progress in Physics*, vol 61 (1998), pp 117-173.

Williams, Colin P. and Clearwater, Scott H. "Explorations in Quantum Computing," New York: Springer-Verlag, 1998.

9 Glossary

This is a glossary of terms and variables used throughout this paper.

λ: In the context of Shor's algorithm in integer such that $m = \frac{\lambda q}{r}$.

a: An argument to the function $\mathcal{F}(a) = x^a \bmod n$. It may be a single integer, or it may denote a superposition of states.

Binary: The base two number system. For more information see Appendix A.

Bit: A thing which can store one item of information, either a 1 or a 0.

Church-Turing Thesis, the hypothesis that:

> Every 'function which would naturally be regarded as computable'
> can be computed by the universal Turing machine.

Classical computer: A computer whose internal workings behave in manner consistent with classical physics. Data registers in a classical computer can not exist in a superposition of states.

Classical physics: The model that was used to describe physical phenomenon before the advent of quantum physics. The predictions of classical physics with regard to the behavior of fundamental particles are incorrect.

Collapse: How the state vector of a quantum mechanical system changes when that system is observed or measured. Since the system can only be measured to be in one of its base states, the state vector will collapse from some superposition of base states into the measured state only.

Complex number: A number of the form $a + i * b$, where a and b are real numbers and i is defined to be the square root of negative one.

Complex vector space: A vector space in which the coordinates of a vector are complex numbers.

Complexity class: A grouping of algorithms based on how their memory usage and number of operations scale with the size of the input.

Coprime: Integers a and b are coprime if their greatest common denominator is one.

Discrete Fourier Transform: In Shor's algorithm this numerical method is used to calculate the multiple of the inverse period, which enables Shor's algorithm to find factors of a number n.

Exponential: A function which grows as: $\mathcal{F}(x) = a^x$, where a is some constant.

Exponential time: This is an attribute of an algorithm which means the number of operations required to compute the answer grows exponentially with the size of the input.

gcd: This is an abbreviation for the mathematical function which calculates the greatest common denominator of two integers. The greatest common denominator of two integers a and b is the largest integer c such that a/c and b/c are integers.

Grover's Search Algorithm: A algorithm designed by L. K. Grover of Bell Labs which finds a element in an unsorted database of size n in $O(\sqrt{n})$ operations on a quantum computer.

Hilbert Space: A complex linear vector space. The complete state of a n state quantum mechanical system can be represented by a vector in an n dimensional Hilbert Space.

i: The square root of -1.

Liner vector space: A vector space in which a vector which is added to or multiplied by another vector results in a vector which lies within the vector space.

Memory register: A array of memory bits, a register of size n may store one of 2^n values.

Mutually perpendicular: In the context of vector spaces mutually perpendicular vectors are vectors such that no one can be decomposed into components of the others.

n: In the context of Shor's algorithm, a number to be factored.

Periodic function: A function with a period r such that $\mathcal{F}(x) = \mathcal{F}(x + r) = \mathcal{F}(x + 2r)$ and so on. Sine and Cosine are periodic functions.

Polynomial time: This is an attribute of an algorithm meaning that the number of operations required to compute the answer grows polynomially with the size of the input.

q: In the context of Shor's algorithm the power of 2 such that $n^2 \leq q < 2n^2$.

Quantum memory register: A array of n qubits which can exist in any superposition of its 2^n base states.

Quantum parallelism: The ability of a quantum computer to perform an operation on a quantum memory register which results in the simultaneous calculation of a function on all superposed values in the quantum memory register.

Quantum physics: Currently the most complete model for describing the behavior of physical systems.

Qubit: A two state quantum mechanical system, which can exist in any superposition of the 0 and 1 state. In this paper I have considered a spin-1/2 particle as a possible candidate for a qubit's physical implementation.

r: In the context of Shor's algorithm the period of the periodic function $x^a \bmod n$.

Shor's Algorithm: An algorithm designed by Peter Shor of Bell Labs which finds factors of a number n in polynomial time on a quantum computer.

Spin-1/2 particle: A fundamental particle, by which I mean it has no components, which can be characterized as having a spin of +1/2 or -1/2.

State vector: The vector in a Hilbert Space which completely describes a quantum mechanical state vector.

Superposition: A mixture of base states. The state vector for a quantum mechanical systems, which can be measured in one of n base states, can in general exist in any combination of components of the base states.

Turing machine: A theoretical computing device consisting of an infinite tape divided into cells which can hold a 1, a 0, or a blank and a head which can move around the tape, read and write bits, and change its own internal state. The Church-Turing Thesis hypothesizes that any computation which can be done on a classical computer can be done on a Turing machine.

Unit vector: A vector whose length is 1.

x: In the context of Shor's algorithm a integer which is coprime to n and used in the function $\mathcal{F}(a) = x^a \bmod n$.

A Mathematics Used in this Paper

This appendix will review some of the mathematics used in the paper that you may not be familiar with. The sections are not intended to be comprehensive for their topic, they only cover what is needed to understand this paper.

A.1 Binary Representation of Numbers

We commonly represent number is base 10, there are 10 elements in our base 10 numbering system, $0, 1, 2, 3, 4, 5, 6, 7, 8$, and 9. In a base n counting system there are n distinct elements, 0 through $n-1$.

When a number which is greater than $n-1$ needs to be displayed in base n it is represented by a string composed of the $n-1$ elements. The value of any given symbol in the string is found by multiplying that symbol by n^x, where x is the number of symbols in the string that are to the right of the symbol in question.

For example; in base 10 the number 982 is equal to $9*10^2+8*10^1+2*10^0$.

In base two the number 10101001 is equal to $1*2^7+0*2^6+1*2^5+0*2^4+1*2^3+0*2^2+0*2^1+1*2^0 = 169$ in base 10.

A.2 Complex Numbers

A complex number is a number of the form $a + i * b$, where a and b are real numbers, and i is defined to be the square root of negative one. Addition of two complex numbers c_1 and c_2 is defined to be:

$$c_1 = a_1 + i * b_1$$

$$c_2 = a_2 + i * b_2$$

$$c_1 + c_2 = a_1 + a_2 + i * (b_1 + b_2)$$

The complex conjugate of a complex number c, denoted c^* is defined to be:

$$c = a + i * b$$

$$c^* = a - i * b$$

Multiplication of two complex numbers c_1 and c_2 is defined to be:

$$c_1 = a_1 + i * b_1$$

$$c_2 = a_2 + i * b_2$$

$$c_1 * c_2 = a_1 * a_2 - b_1 * b_2 + i * (a_1 * b_2 + a_2 * b_1)$$

Euler's Formula for complex numbers states that $e^{ix} = \cos x + i * \sin x$, this relationship is used in the discrete Fourier transform of Shor's algorithm.

A.3 Vector Mathematics

The only vector operations that are used in our simulation of Shor's algorithm are addition, length determination, and scaling. The vector in question represents the state vector of a quantum mechanical system; a complex vector in a Hilbert Space. The vector can be represented by projections of the vector onto each of the perpendicular base vectors which define the Hilbert Space.

For example, an n state quantum system requires a n dimensional Hilbert Space to represent its state vector. The quantum system can be measured in any of the n states, and to represent this we imagine each of the n states as mutually perpendicular axes within our Hilbert space. Thus the state vector for a system in the j'th state is equal to:

$$\begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix}$$

For the n states, where the number at the top of the column is the length of the state vector projected onto the 1st state, and the 1 appears in the j'th row.

To add two vectors we simply add their components.

$$\begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} a_1 + b_1 \\ a_2 + b_2 \\ a_3 + b_3 \end{pmatrix}$$

Since this vector lies in a Hilbert Space the projections of the state vector onto the coordinate axes are allowed to be complex numbers, thus the definition of length is slightly different from what is expected.

The length of a vector in a Hilbert space with n components is defined to be: $\sqrt{\sum_{j=1}^{n} |w_j|^2}$ where w_j is the value of the j'th component of the vector, and $|w_j|^2$ is defined to be w_j times its complex conjugate, or when $w_j = a + i * b$, $|w_j|^2 = a^2 + b^2$.

To scale a vector by any length l you simply multiply each component of the vector by the value l. In particular to scare a vector to length 1 you multiply each component by the inverse length of the vector.

B Actual Quantum Computers and Further Research

There is great interest in building an actual quantum computer. Many strategies for building a quantum memory register are currently being evaluated. The study of quantum computing is a relatively new field, and the experimental implementation is still in its infancy. Consequently the world wide web is the best place to search for information regarding current research.

One possible way to implement a quantum memory register is by using nuclear magnetic resonance techniques to manipulate classical fluids to perform quantum calculations. To read more on this see:

http://www.sciam.com/1998/0698issue/0698gershenfeld.html

http://squint.stanford.edu/qc/overview.html

A second approach is to use ion traps and lasers to hold atomic ions in place. To read more about this method check:

http://physics.colorado.edu/faculty/monroe.c.html

http://www.bldrdoc.gov/timefreq/ion/index.htm

A good place to look for original research papers is the Los Alamos National Laboratories E-print archive at:

http://quickreviews.org/

This is by no means a comprehensive list of approaches to or research regarding quantum computing. Given the dynamic nature of the web, you will be able find more current information about quantum computing from your favorite web search engine or directory service.

C Other Quantum Computer Simulators

The following web sites have information about quantum computing, and have code for simulating the action of a quantum computer. I found these sites to be great resources during my study of quantum computing.

http://tph.tuwien.ac.at/ oemer/qc/qcl/qcl.html

http://tph.tuwien.ac.at/ oemer/prakt/prakt.html

http://www.openqubit.org/

D Code for my Simulation of Shor's Algorithm

Here is the source code for my simulation of Shor's algorithm. I have tried to document the code as well as possible, buy if you find something difficult please write me at mjhayward@google.com and I will attempt to explain it as well as I can. The code was compiled with g++ version 2.8.1, on a Pentium Pro 200 running Linux 2.0.33. I have also compiled it on a UltraSparc running SunOS 5.6 also using g++ 2.8.1, with one modification. I hope that you will find it easy to compile the code with g++. If you have trouble compiling please send me information about your compiler, OS, and error messages, and I will see what I can do.

D.1 complex.C

```
#include <iostream.h>
#include <math.h>
#include <stdlib.h>

class Complex {
public:
  Complex();                  //Default constructor
  ~Complex();                 //Default destructor.
  void Set(double new_real, double new_imaginary); //Set data members.
  double Real();              //Return the real part.
  double Imaginary();         //Return the imaginary part.
  Complex operator+(Complex); //Overloaded + operator
  Complex operator*(Complex); //Overloaded * operator
  Complex operator=(Complex); //Overloaded = operator
  int operator==(Complex);    //Overloaded == operator
private:
  double real;
  double imaginary;
};

//Complex constructor, initialises to 0 + i0.
Complex::Complex() {
  real = 0;
  imaginary = 0;
}

//Complex destructor.
Complex::~Complex() {};

//Overloaded = operator.
Complex Complex::operator=(Complex c) {
  if (&c != this) {
    real = c.Real();
    imaginary = c.Imaginary();
    return *this;
  }
}

//Overloaded + operator.
Complex Complex::operator+(Complex c) {
  Complex tmp;
  double new_real, new_imaginary;
  new_real = real + c.Real();
```

```
  new_imaginary = imaginary + c.Imaginary();
  tmp.Set(new_real,new_imaginary);
  return tmp;
}

//Overloaded * operator.
Complex Complex::operator*(Complex c) {
  Complex tmp;
  double new_real, new_imaginary;
  new_real = real * c.Real() - imaginary * c.Imaginary();
  new_imaginary = real * c.Imaginary() + imaginary * c.Real();
  tmp.Set(new_real,new_imaginary);
  return tmp;
}

//Overloaded == operator.  Small error tolerances.
int Complex::operator==(Complex c) {
  //This is to take care of round off errors.
  if (fabs(c.Real() - real) > pow(10,-14)) {
    return 0;
  }
  if (fabs(c.Imaginary()- imaginary) > pow(10,-14)) {
    return 0;
  }
  return 1;
}

//Sets private data members.
void Complex::Set(double new_real, double new_imaginary) {
  real = new_real;
  imaginary = new_imaginary;
}

//Returns the real part of the complex number.
double Complex::Real() {
  return real;
}

//Returns the imaginary part of the complex number.
double Complex::Imaginary() {
  return imaginary;
}
```

D.2 qureg.C

```
#include <iostream.h>
#include <math.h>
#include <time.h>

class QuReg {
public:
  //Default constructor.  Size is the size in bits of our register.
  //In our implementation of Shor's algorithm we will need size bits
  //to represent our value for "q" which is a number we have chosen
  //with small prime factors which is between 2n^2 and 3n^2 inclusive
  //where n is the number we are trying to factor.  We envision our the
  //description of our register of size "S" as 2^S complex number,
  //representing the probability of finding the register on one of or
  //2^S base states.  Thus we use an array of size 2^S, of Complex
  //numbers.  Thus if the size of our register is 3 bits array[7] is
  //the probability amplitude of the state |1,1,1>, and array[7] *
  //Complex Conjugate(array[7]) = probability of choosing that state.
  //We use normalised state vectors thought the simulation, thus the
  //sum of all possible states times their complex conjugates is = 1.
  QuReg(int size);

  QuReg(); //Default Constructor

  QuReg(const QuReg &); //Copy constructor

  ~QuReg(); //Default destructor.

  int DecMeasure(); //Measures our quantum register, and returns the
  //decimal interpretation of the bitstring measured.

  //Dumps all the information about the quantum register.  This has no
  //physical reality, it is only there for debugging.  When verbose !=
  //0 we return every value, when verbose = 0 we return only
  //probability amplitudes which differ from 0.
  void Dump(int verbose);

  //Sets state of the qubits using the arrays of complex amplitudes.
  void SetState(Complex new_state[]);

  //Sets the state to an equal superposition of all possible states
  //between 0 and number inclusive.
  void SetAverage(int number);

  //Normalise the state amplitudes.
```

```
  void Norm();

  //Get the probability of a given state.  This is used in the
  //discrete Fourier transformation.  In a real quantum computer such
  //an operation would not be possible, on the flip side, it would
  //also not be necessary as you could simply build a DFT gate, and
  //run your superposition through it to get the right answer.
  Complex GetProb(int state);

  //Return the size of the register.
  int Size();

private:
  int reg_size;
  Complex *State;
};

QuReg::QuReg() {
  reg_size = 0;
  State = 0;
}

//Constructor.
QuReg::QuReg(int size) {
  reg_size = size;
  State = new Complex[(int)pow(2, reg_size)];
  srand(time(NULL));
}

//Copy Constructor
QuReg::QuReg(const QuReg & old) {
  reg_size = old.reg_size;
  int reg_length = (int) pow(2, reg_size);
  State = new Complex[reg_length];
  for (int i = 0 ; i < reg_length ; i++) {
    State[i] = old.State[i];
  }
}

//Destructor.
QuReg::~QuReg() {
  delete [] State;
}

//Return the probability amplitude of the state'th state.
Complex QuReg::GetProb(int state) {
```

```
  if (state >= pow(2, reg_size)) {
    cout << "You are trying to measure past the end of an array in qureg::GetProb!"
 << endl << flush;
  } else {
    return(State[state]);
  }
}

//Normalise the probability amplitude, this ensures that the sum of
//the sum of the squares of all the real and imaginary components is
//equal to one.
void QuReg::Norm() {
  double b;
  double f, g;
  b = 0;
  for (int i = 0; i < pow(2, reg_size) ; i++) {
    b += pow(State[i].Real(), 2) + pow(State[i].Imaginary(), 2);
  }
  b = pow(b, -.5);
  for (int i = 0; i < pow(2, reg_size) ; i++) {
    f = State[i].Real() * b;
    g = State[i].Imaginary() * b;
    State[i].Set(f, g);
  }
}

//Returns the size of the register.
int QuReg::Size() {
  return reg_size;
}

//Measure a state, and return the decimal value measured.  Collapse
//the state so that the probability of measuring the measured value in
//the future is 1, and the probability of measuring any other state is
//0.
int QuReg::DecMeasure() {
  int done = 0;
  int DecVal = -1; //-1 is an error, we did not measure anything.
  double rand1, a, b;
  rand1 = rand()/(double)RAND_MAX;
  a = b = 0;
  for (int i = 0 ; i < pow(2, reg_size)  ;i++) {
    if (!done ){
      b += pow(State[i].Real(), 2) + pow(State[i].Imaginary(), 2);
      if (b > rand1 && rand1 > a) {
//We have just measured the i state.
```

```
for (int j = 0; j < pow(2, reg_size) ; j++) {
  State[j].Set(0,0);
}
State[i].Set(1,0);
DecVal = i;
done = 1;
      }
      a += pow(State[i].Real(), 2) + pow(State[i].Imaginary(), 2);
    }
  }
  return DecVal;
}

//For debugging, output information about the register.
void QuReg::Dump(int verbose) {
  for (int i = 0 ; i < pow(2, reg_size) ; i++) {
    if (verbose || fabs(State[i].Real()) > pow(10,-14)
|| fabs(State[i].Imaginary()) > pow(10,-14)) {
      cout << "State " << i << " has probability amplitude "
   << State[i].Real() << " + i" << State[i].Imaginary()
   << endl << flush;
    }
  }
}

//Set the states to those given in the new_state array.
void QuReg::SetState(Complex new_state[]) {
  //Beware, this function will cause segfaults if new_state is too
  //small.
  for (int i = 0 ; i < pow(2, reg_size) ; i++) {
    State[i].Set(new_state[i].Real(), new_state[i].Imaginary());
  }
}

//Set the State to an equal superposition of the integers 0 -> number
//- 1
void QuReg::SetAverage(int number) {
  if (number >= pow(2, reg_size)) {
    cout << "Error, initialising past end of array in qureg::SetAverage.\n";
  } else {
    double prob;
    prob = pow(number, -.5);
    for (int i = 0 ; i <= number ; i++) {
      State[i].Set(prob, 0);
    }
  }
```

```
}
```

D.3 shor.C

```
#include <iostream.h>
#include <math.h>
#include <time.h>
#include "complex.C"
#include "util.C"

int main() {
  //Establish a random seed.
  srand(time(NULL));

  //Output standard greeting.
  cout << "Welcome to the simulation of Shor's algorithm." << endl
       << "There are four restrictions for Shor's algorithm:" << endl
       << "1) The number to be factored must be >= 15." << endl
       << "2) The number to be factored must be odd." << endl
       << "3) The number must not be prime." << endl
       << "4) The number must not be a prime power." << endl
       << endl << "There are efficient classical methods of factoring "
       << "any of the above numbers, or determining that they are prime."
       << endl << endl << "Input the number you wish to factor." << endl
       << flush;

  //n is the number we are going to factor, get n.
  int n;
  cin >> n;

  //Test to see if n is factorable by Shor's algorithm.
  //Exit if the number is even.
  if (n%2 == 0) {
    cout << "Error, the number must be odd!" << endl << flush;
    exit(0);
  }
  //Exit if the number is prime.
  if (TestPrime(n)) {
    cout << "Error, the number must not be prime!" << endl << flush;
    exit(0);
  }
  //Prime powers are prime numbers raised to integral powers.
  //Exit if the number is a prime power.
  if (TestPrimePower(n)) {
    cout << "Error, the number must not be a prime power!" << endl << flush;
    exit(0);
  }
```

```
//Now we must pick a random integer x, coprime to n.  Numbers are
//coprime when their greatest common denominator is one.  One is not
//a useful number for the algorithm.
int x = 0;
x = 1+ (int)((n-1)*(double)rand()/(double)RAND_MAX);
while (GCD(n,x) != 1 || x == 1) {
    x = 1 + (int)((n-1)*(double)rand()/(double)RAND_MAX);
}
cout << "Found x to be " << x << "." << endl << flush;

//Now we must figure out how big a quantum register we need for our
//input, n.  We must establish a quantum register big enough to hold
//an equal superposition of all integers 0 through q - 1 where q is
//the power of two such that n^2 <= q < 2n^2.
int q;
q = GetQ(n);
cout << "Found q to be " << q << "." << endl << flush;

//Create the register.
QuReg * reg1 = new QuReg(RegSize(q) - 1);
cout << "Made register 1 with register size = " << RegSize(q) << endl
     << flush;

//This array will remember what values of q produced for x^q mod n.
//It is necessary to retain these values for use when we collapse
//register one after measuring register two.  In a real quantum
//computer these registers would be entangled, and thus this extra
//bookkeeping would not be needed at all.  The laws of quantum
//mechanics dictate that register one would collapse as well, and
//into a state consistent with the measured value in resister two.
int * modex = new int[q];

//This array holds the probability amplitudes of the collapsed state
//of register one, after register two has been measured it is used
//to put register one in a state consistent with that measured in
//register two.
Complex *collapse = new Complex[q];

//This is a temporary value.
Complex tmp;

//This is a new array of probability amplitudes for our second
//quantum register, that populated by the results of x^a mod n.
Complex *mdx = new Complex[(int)pow(2,RegSize(n))];

// This is the second register.  It needs to be big enough to hold
```

```
// the superposition of numbers ranging from 0 -> n - 1.
QuReg *reg2 = new QuReg(RegSize(n));
cout << "Created register 2 of size " << RegSize(n) << endl << flush;

//This is a temporary value.
int tmpval;

//This is a temporary value.
int value;

//c is some multiple lambda of q/r, where q is q in this program,
//and r is the period we are trying to find to factor n.  m is the
//value we measure from register one after the Fourier
//transformation.
double c,m;

//This is used to store the denominator of the fraction p / den where
//p / den is the best approximation to c with den <= q.
int den;

//This is used to store the numerator of the fraction p / den where
//p / den is the best approximation to c with den <= q.
int p;

//The integers e, a, and b are used in the end of the program when
//we attempts to calculate the factors of n given the period it
//measured.
//Factor is the factor that we find.
int e,a,b, factor;

//Shor's algorithm can sometimes fail, in which case you do it
//again.  The done variable is set to 0 when the algorithm has
//failed.  Only try a maximum number of tries.
int done = 0;
int tries = 0;
while (!done) {
  if (tries >= 5) {
    cout << "There have been five failures, giving up." << endl << flush;
    exit(0);
  }
  //Now populate register one in an even superposition of the
  //integers 0 -> q - 1.
  reg1->SetAverage(q - 1);

  //Now we preform a modular exponentiation on the superposed
  //elements of reg 1.  That is, perform x^a mod n, but exploiting
```

```
    //quantum parallelism a quantum computer could do this in one
    //step, whereas we must calculate it once for each possible
    //measurable value in register one.  We store the result in a new
    //register, reg2, which is entangled with the first register.
    //This means that when one is measured, and collapses into a base
    //state, the other register must collapse into a superposition of
    //states consistent with the measured value in the other..  The
    //size of the result modular exponentiation will be at most n, so
    //the number of bits we will need is therefore less than or equal
    //to log2 of n.  At this point we also maintain a array of what
    //each state produced when modularly exponised, this is because
    //these registers would actually be entangled in a real quantum
    //computer, this information is needed when collapsing the first
    //register later.

    //This counter variable is used to increase our probability amplitude.
    tmp.Set(1,0);

    //This for loop ranges over q, and puts the value of x^a mod n in
    //modex[a].  It also increases the probability amplitude of the value
    //of mdx[x^a mod n] in our array of complex probabilities.
    for (int i = 0 ; i < q ; i++) {
      //We must use this version of modexp instead of c++ builtins as
      //they overflow when x^i > 2^31.
      tmpval = modexp(x,i,n);
      modex[i] = tmpval;
      mdx[tmpval] = mdx[tmpval] + tmp;
    }

    //Set the state of register two to what we calculated it should be.
    reg2->SetState(mdx);

    //Normalise register two, so that the probability of measuring a
    //state is given by summing the squares of its probability
    //amplitude.
    reg2->Norm();

    //Now we measure reg1.
    value = reg2->DecMeasure();

    //Now we must using the information in the array modex collapse
    //the state of register one into a state consistent with the value
    //we measured in register two.
    for (int i = 0 ; i < q ; i++) {
      if (modex[i] == value) {
collapse[i].Set(1,0);
```

```
      } else {
collapse[i].Set(0,0);
      }
    }

    //Now we set the state of register one to be consistent with what
    //we measured in state two, and normalise the probability
    //amplitudes.
    reg1->SetState(collapse);
    reg1->Norm();

    //Here we do our Fourier transformation.
    cout << "Begin Discrete Fourier Transformation!" << endl << flush;
    DFT(reg1, q);

    //Next we measure register one, due to the Fourier transform the
    //number we measure, m will be some multiple of lambda/r, where
    //lambda is an integer and r is the desired period.
    m = reg1->DecMeasure();

    //If nothing goes wrong from here on out we are done.
    done = 1;

    //If we measured zero, we have gained no new information about the
    //period, we must try again.
    if (m == 0) {
      cout << "Measured, 0 this trial a failure!" << endl << flush;
      done = 0;
    }

    //The DecMeasure subroutine will return -1 as an error code, due
    //to rounding errors it will occasionally fail to measure a state.
    if (m == -1) {
      cout << "We failed to measure anything, this trial a failure!"
   << "  Trying again." << endl << flush;
      done = 0;
    }

    //If nothing has gone wrong, try to determine the period of our
    //function, and get factors of n.
    if (done) {
      //Now c =~ lambda / r for some integer lambda.  Borrowed with
      //modifications from Berhnard Ohpner.
      c = (double)m  / (double)q;

      //Calculate the denominator of the best rational approximation
```

```
      //to c with den < q.  Since c is lambda / r for some integer
      //lambda, this will provide us with our guess for r, our period.
      den = denominator(c, q);

      //Calculate the numerator from the denominator.
      p = (int)floor(den * c + 0.5);

      //Give user information.
      cout << "measured " << m <<", approximation for " << c << " is "
   << p << " / " << den << endl << flush;

      //The denominator is our period, and an odd period is not
      //useful as a result of Shor's algorithm.  If the denominator
      //times two is still less than q we can use that.
      if (den % 2 == 1 && 2 * den < q ){
cout << "Odd denominator, expanding by 2\n";
p = 2 * p;
den = 2 * den;
      }

      //Initialise helper variables.
      e = a = b = factor = 0;

      // Failed if odd denominator.
      if (den % 2 == 1)  {
cout <<  "Odd period found.  This trial failed."
     << "  Trying again." << endl << flush;
done = 0;
      } else {
//Calculate candidates for possible common factors with n.
cout <<  "possible period is " << den << endl << flush;
e = modexp(x, den / 2, n);
a = (e + 1) % n;
b = (e + n - 1) % n;
cout << x << "^" << den / 2 << " + 1 mod " << n << " = " << a
     << "," << endl
     << x << "^" << den / 2 << " - 1 mod " << n << " = " << b
     << endl << flush;
factor = max(GCD(n,a),GCD(n,b));
      }
    }

    //GCD will return a -1 if it tried to calculate the GCD of two
    //numbers where at some point it tries to take the modulus of a
    //number and 0.
    if (factor == -1) {
```

```
      cout << "Error, tried to calculate n mod 0 for some n.  Trying again."
   << endl << flush;
      done = 0;
    }

    if ((factor == n || factor == 1) && done == 1) {
      cout << "Found trivial factors 1 and " << n
   << ".  Trying again." << endl << flush;
      done = 0;
    }

    //If nothing else has gone wrong, and we got a factor we are
    //finished.  Otherwise start over.
    if (factor != 0 && done == 1) {
      cout << n << " = " << factor << " * " << n / factor << endl << flush;
    } else if (done == 1) {
      cout << "Found factor to be 0, error.  Trying again." << endl
   << flush;
      done = 0;
    }
    tries++;
  }
  delete reg1;
  delete reg2;
  delete [] modex;
  delete [] collapse;
  delete [] mdx;
  return 1;
}
```

D.4 util.C

```
#include <iostream.h>
#include <math.h>
#include "qureg.C"
#define PI 3.14159265359

//This function takes an integer input and returns 1 if it is a prime
//number, and 0 otherwise.
int TestPrime(int n) {
  int i;
  for (i = 2 ; i <= floor(sqrt(n)) ; i++) {
    if (n % i == 0) {
      return(0);
    }
  }
  return(1);
}

//This function takes an integer input and returns 1 if it is equal to a
//prime number raised to an integer power, and 0 otherwise.
int TestPrimePower(int n) {
  int i,j;
  j = 0;
  i = 2;
  while ((i <= floor(pow(n, .5))) && (j == 0)) {
    if((n % i) == 0) {
      j = i;
    }
    i++;
  }
  for (int i = 2 ; i <= (floor(log(n) / log(j)) + 1) ; i++) {
    if(pow(j , i) == n) {
      return(1);
    }
  }
  return(0);
}

//This function computes the greatest common denominator of two integers.
//Since the modulus of a number mod 0 is not defined, we return a -1 as
//an error code if we ever would try to take the modulus of something and
//zero.
int GCD(int a, int b) {
  int d;
  if (b != 0) {
```

```
    while (a % b != 0) {
      d = a % b;
      a = b;
      b = d;
    }
  } else {
    return -1;
  }
  return(b);
}

//This function takes and integer argument, and returns the size in bits
//needed to represent that integer.
int RegSize(int a) {
  int size = 0;
  while(a != 0) {
    a = a>>1;
    size++;
  }
  return(size);
}


//q is the power of two such that n^2 <= q < 2n^2.
int GetQ(int n) {
  int power = 8; //265 is the smallest q ever is.
  while (pow(2,power) < pow(n,2)) {
    power = power + 1;
  }
  return((int)pow(2,power));
}

//This function takes three integers, x, a, and n, and returns x^a mod n.
//This algorithm is known as the "Russian peasant method," I belive.
int modexp(int x, int a, int n) {
  int value = 1;
  int tmp;
  tmp = x % n;
  while (a > 0) {
    if (a & 1) {
      value = (value * tmp) % n;
    }
    tmp = tmp * tmp % n;
    a = a>>1;
  }
  return value;
```

```
}


// This function finds the denominator q of the best rational
// denominator q for approximating p / q for c with q < qmax.
int denominator(double c, int qmax) {
  double y = c;
  double z;
  int q0 = 0;
  int q1 = 1;
  int q2 = 0;
  while (1) {
    z = y - floor(y);
    if (z < 0.5 / pow(qmax,2)) {
      return(q1);
    }
    if (z != 0) {
      //Can't divide by 0.
      y = 1 / z;
    } else {
      //Warning this is broken if q1 == 0, but that should never happen.
      return(q1);
    }
    q2 = (int)floor(y) * q1 + q0;
    if (q2 >= qmax) {
      return(q1);
    }
    q0 = q1;
    q1 = q2;
  }
}

//This function takes two integer arguments and returns the greater of
//the two.
int max(int a, int b) {
  if (a > b) {
    return(a);
  }
  return(b);
}

//This function computes the discrete Fourier transformation on a register's
//0 -> q - 1 entries.
void DFT(QuReg * reg, int q) {
  //The Fourier transform maps functions in the time domain to
  //functions in the frequency domain.  Frequency is 1/period, thus
```

```
  //this Fourier transform will take our periodic register, and peak it
  //at multiples of the inverse period.  Our Fourier transformation on
  //the state a takes it to the state: q^(-.5) * Sum[c = 0 -> c = q - 1,
  //c * e^(2*Pi*i*a*c / q)].  Remember, e^ix = cos x + i*sin x.

  Complex * init = new Complex[q];
  Complex tmpcomp;
  tmpcomp.Set(0,0);

  //Here we do things that a real quantum computer couldn't do, such
  //as look as individual values without collapsing state.  The good
  //news is that in a real quantum computer you could build a gate
  //which would what this out all in one step.

  int count = 0;
  double tmpreal = 0;
  double tmpimag = 0;
  double tmpprob = 0;
  for (int a = 0 ; a < q ; a++) {
    //This if statement helps prevent previos round off errors from
    //propogating further.
    if ((pow(reg->GetProb(a).Real(),2) +
 pow(reg->GetProb(a).Imaginary(),2)) > pow(10,-14)) {
      for (int c = 0 ; c < q ; c++) {
tmpcomp.Set(pow(q,-.5) * cos(2*PI*a*c/q),
    pow(q,-.5) * sin(2*PI*a*c/q));
init[c] = init[c] + (reg->GetProb(a) * tmpcomp);
      }
    }
    count++;
    if (count == 100) {
      cout << "Making progress in Fourier transform, "
   << 100*((double)a / (double)(q - 1)) << "% done!"
   << endl << flush;
      count = 0;
    }
  }
  reg->SetState(init);
  reg->Norm();
  delete [] init;
}
```

D.5 qubit.cpp

Below is the much simpler code for a single qubit. You may find it entertaining and educational to write a simple driver program for this class to toy with a single qubit.

```
#include <iostream.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include "complex.C"

class Qubit {
public:
  Qubit();            //Default constructor.
  ~Qubit();           //Default destructor.
  int Measure();      //Returns zero_state = 0 or one_state = 1 in accordance
                      //with the probabilities of zero_state and one_state.
  void Dump();        //Prints our zero_state, and one_state, without
                      //disturbing anything, this operation has no physical
                      //realization, it is only for information and debuging.
                      //It should never be used in an algorithm for
                      //information.
  void SetState(Complex zero_prob, Complex one_prob);
                      // Takes two complex numbers and sets the states to
                      //those values.
  void SetAverage();  //Sets the state to 2^(1/2) zero_state, 2^(1/2)
                      //one_state.  No imaginary/phase component.
  double MCC(int state);  //Multiply the zero or one state by its complex
                          //conjugate, and return the value.  This value
                          //will always be a real number, with no imaginary
                          //component.

private:
  Complex zero_state;
  //The probability of finding the Qubit in the zero or all imarinary
  //state.  I currently use only the real portion.

  Complex one_state;
  //The probability of finding the Qubit in the one or all real state.
  //I currently use only the real portion.

  //|zero_state|^2 + |one_state|^2 should always be 1.
  //This notation means z_s * ComplexConjugate(z_s) + o_s *
  //ComplexConjugate(o_s) = 1.
};
```

```
//Qubit constructor, initially sets things in the zero state.
Qubit::Qubit() {
  zero_state.Set(1,0);
  one_state.Set(0,0);
  srand(time(NULL));
}

//Returns <state>_state * ComplexConjugate(<state>_state).
double Qubit::MCC(int state) {
  if (state == 0) {
    return (pow(zero_state.Real(), 2) + pow(zero_state.Imaginary(), 2));
  }
  return (pow(one_state.Real(), 2) + pow(one_state.Imaginary(), 2));
}

//Measurement operator.  Destructively collapses superpositions.
int Qubit::Measure() {
  double rand1;
  rand1 = rand()/(double)RAND_MAX;
  //Assumes that the sum of the squares of the amplitudes are = 1

  if (MCC(0) > rand1) {
    //Should I be collapsing phases as well?
    zero_state.Set(1,0);
    one_state.Set(0,0);
    return 0;
  } else {
    //Should I be collapsing phases as well?
    zero_state.Set(0,0);
    one_state.Set(1,0);
    return 1;
  }
}

//Outputs state info for our qubit.  For debugging purposes.
void Qubit::Dump() {
  cout << "Amplitude of the zero state is "
       << zero_state.Real() << " +i" << zero_state.Imaginary() << endl
       << flush;
  cout << "Amplutide of the one state is "
       << one_state.Real() << " +i" << one_state.Imaginary() << endl
       << flush;
}

//Sets the zero and one states to arbitrary amplitudes.  Outputs
//an error message if the two values MCC'ed != 1 + 0i.
```

```
void Qubit::SetState(Complex zero_prob, Complex one_prob) {
  zero_state = zero_prob;
  one_state = one_prob;
  //Determine if |zero_state|^2 + |one_state|^2 == 1, if not we
  //are not in a valid state.
  double probab;
  probab = MCC(0) + MCC(1);
  if (fabs(probab - 1) > pow(10, -10)) {
    //This funny expression
    //allows us some rounding errors.
    cout << "Warning, total probability for in SetState is different from 1." << endl << flu
  }
}

//Sets the qubit 1/2 way between the 0 state and the 1 state.  No phase.
void Qubit::SetAverage() {
  zero_state.Set(pow(2, -.5), 0);
  one_state.Set(pow(2, -.5), 0);
}
```

E Sample Output

E.1 Sample Output for $n = 17$

```
Welcome to the simulation of Shor's algorithm.
There are four restrictions for Shor's algorithm:
1) The number to be factored must be >= 15.
2) The number to be factored must be odd.
3) The number must not be prime.
4) The number must not be a prime power.

There are efficient classical methods of factoring any of the above
numbers, or determining that they are prime.

Input the number you wish to factor.
17
Error, the number must not be prime!
```

E.2 Sample Output for $n = 15$

```
Welcome to the simulation of Shor's algorithm.
There are four restrictions for Shor's algorithm:
1) The number to be factored must be >= 15.
2) The number to be factored must be odd.
3) The number must not be prime.
4) The number must not be a prime power.

There are efficient classical methods of factoring any of the above
numbers, or determining that they are prime.

Input the number you wish to factor.
15
Found x to be 14.
Found q to be 256.
Made register 1 with register size = 9
Created register 2 of size 4
Begin Discrete Fourier Transformation!
Making progress in Fourier transform, 38.8235% done!
Making progress in Fourier transform, 78.0392% done!
Measured, 0 this trial a failure!
Begin Discrete Fourier Transformation!
Making progress in Fourier transform, 38.8235% done!
Making progress in Fourier transform, 78.0392% done!
Measured, 0 this trial a failure!
Begin Discrete Fourier Transformation!
Making progress in Fourier transform, 38.8235% done!
Making progress in Fourier transform, 78.0392% done!
measured 128, approximation for 0.5 is 1 / 2
possible period is 2
14^1 + 1 mod 15 = 0,
14^1 - 1 mod 15 = 13
Found trivial factors 1 and 15.   Trying again.
Begin Discrete Fourier Transformation!
Making progress in Fourier transform, 38.8235% done!
Making progress in Fourier transform, 78.0392% done!
Measured, 0 this trial a failure!
Begin Discrete Fourier Transformation!
Making progress in Fourier transform, 38.8235% done!
Making progress in Fourier transform, 78.0392% done!
measured 128, approximation for 0.5 is 1 / 2
possible period is 2
14^1 + 1 mod 15 = 0,
14^1 - 1 mod 15 = 13
Found trivial factors 1 and 15.   Trying again.
```

```
There have been five failures, giving up.
```

E.3 Sample Output for $n = 33$

```
Welcome to the simulation of Shor's algorithm.
There are four restrictions for Shor's algorithm:
1) The number to be factored must be >= 15.
2) The number to be factored must be odd.
3) The number must not be prime.
4) The number must not be a prime power.

There are efficient classical methods of factoring any of the above
numbers, or determining that they are prime.

Input the number you wish to factor.
33
Found x to be 5.
Found q to be 2048.
Made register 1 with register size = 12
Created register 2 of size 6
Begin Discrete Fourier Transformation!
Making progress in Fourier transform, 4.83635% done!
Making progress in Fourier transform, 9.72154% done!
Making progress in Fourier transform, 14.6067% done!
Making progress in Fourier transform, 19.4919% done!
Making progress in Fourier transform, 24.3771% done!
Making progress in Fourier transform, 29.2623% done!
Making progress in Fourier transform, 34.1475% done!
Making progress in Fourier transform, 39.0327% done!
Making progress in Fourier transform, 43.9179% done!
Making progress in Fourier transform, 48.8031% done!
Making progress in Fourier transform, 53.6883% done!
Making progress in Fourier transform, 58.5735% done!
Making progress in Fourier transform, 63.4587% done!
Making progress in Fourier transform, 68.3439% done!
Making progress in Fourier transform, 73.2291% done!
Making progress in Fourier transform, 78.1143% done!
Making progress in Fourier transform, 82.9995% done!
Making progress in Fourier transform, 87.8847% done!
Making progress in Fourier transform, 92.7699% done!
Making progress in Fourier transform, 97.6551% done!
measured 1843, approximation for 0.449951 is 463 / 1029
Odd period found.   This trial failed. Trying again.
Begin Discrete Fourier Transformation!
Making progress in Fourier transform, 4.83635% done!
Making progress in Fourier transform, 9.72154% done!
Making progress in Fourier transform, 14.6067% done!
Making progress in Fourier transform, 19.4919% done!
```

```
Making progress in Fourier transform, 24.3771% done!
Making progress in Fourier transform, 29.2623% done!
Making progress in Fourier transform, 34.1475% done!
Making progress in Fourier transform, 39.0327% done!
Making progress in Fourier transform, 43.9179% done!
Making progress in Fourier transform, 48.8031% done!
Making progress in Fourier transform, 53.6883% done!
Making progress in Fourier transform, 58.5735% done!
Making progress in Fourier transform, 63.4587% done!
Making progress in Fourier transform, 68.3439% done!
Making progress in Fourier transform, 73.2291% done!
Making progress in Fourier transform, 78.1143% done!
Making progress in Fourier transform, 82.9995% done!
Making progress in Fourier transform, 87.8847% done!
Making progress in Fourier transform, 92.7699% done!
Making progress in Fourier transform, 97.6551% done!
measured 1024, approximation for 0.25 is 1 / 4
possible period is 4
5^2 + 1 mod 33 = 26,
5^2 - 1 mod 33 = 24
33 = 3 * 11
```

E.4 Sample Output for $n = 129$

```
Welcome to the simulation of Shor's algorithm.
There are four restrictions for Shor's algorithm:
1) The number to be factored must be >= 15.
2) The number to be factored must be odd.
3) The number must not be prime.
4) The number must not be a prime power.

There are efficient classical methods of factoring any of the above
numbers, or determining that they are prime.

Input the number you wish to factor.
129
Found x to be 83.
Found q to be 32768.
Made register 1 with register size = 16
Created register 2 of size 8
Begin Discrete Fourier Transformation!
Making progress in Fourier transform, 0.302133% done!
Making progress in Fourier transform, 0.607318% done!
Making progress in Fourier transform, 0.912503% done!
Making progress in Fourier transform, 1.21769% done!
Making progress in Fourier transform, 1.52287% done!
Making progress in Fourier transform, 1.82806% done!
Making progress in Fourier transform, 2.13324% done!
Making progress in Fourier transform, 2.43843% done!
Making progress in Fourier transform, 2.74361% done!
Making progress in Fourier transform, 3.0488% done!
Making progress in Fourier transform, 3.35398% done!
Making progress in Fourier transform, 3.65917% done!
Making progress in Fourier transform, 3.96435% done!
Making progress in Fourier transform, 4.26954% done!
Making progress in Fourier transform, 4.57472% done!
Making progress in Fourier transform, 4.87991% done!
Making progress in Fourier transform, 5.18509% done!
Making progress in Fourier transform, 5.49028% done!
Making progress in Fourier transform, 5.79546% done!
Making progress in Fourier transform, 6.10065% done!
Making progress in Fourier transform, 6.40584% done!
Making progress in Fourier transform, 6.71102% done!
Making progress in Fourier transform, 7.01621% done!
Making progress in Fourier transform, 7.32139% done!
Making progress in Fourier transform, 7.62658% done!
Making progress in Fourier transform, 7.93176% done!
Making progress in Fourier transform, 8.23695% done!
```

```
Making progress in Fourier transform, 8.54213% done!
Making progress in Fourier transform, 8.84732% done!
Making progress in Fourier transform, 9.1525% done!
Making progress in Fourier transform, 9.45769% done!
Making progress in Fourier transform, 9.76287% done!
Making progress in Fourier transform, 10.0681% done!
Making progress in Fourier transform, 10.3732% done!
Making progress in Fourier transform, 10.6784% done!
Making progress in Fourier transform, 10.9836% done!
Making progress in Fourier transform, 11.2888% done!
Making progress in Fourier transform, 11.594% done!
Making progress in Fourier transform, 11.8992% done!
Making progress in Fourier transform, 12.2044% done!
Making progress in Fourier transform, 12.5095% done!
Making progress in Fourier transform, 12.8147% done!
Making progress in Fourier transform, 13.1199% done!
Making progress in Fourier transform, 13.4251% done!
Making progress in Fourier transform, 13.7303% done!
Making progress in Fourier transform, 14.0355% done!
Making progress in Fourier transform, 14.3406% done!
Making progress in Fourier transform, 14.6458% done!
Making progress in Fourier transform, 14.951% done!
Making progress in Fourier transform, 15.2562% done!
Making progress in Fourier transform, 15.5614% done!
Making progress in Fourier transform, 15.8666% done!
Making progress in Fourier transform, 16.1718% done!
Making progress in Fourier transform, 16.4769% done!
Making progress in Fourier transform, 16.7821% done!
Making progress in Fourier transform, 17.0873% done!
Making progress in Fourier transform, 17.3925% done!
Making progress in Fourier transform, 17.6977% done!
Making progress in Fourier transform, 18.0029% done!
Making progress in Fourier transform, 18.3081% done!
Making progress in Fourier transform, 18.6132% done!
Making progress in Fourier transform, 18.9184% done!
Making progress in Fourier transform, 19.2236% done!
Making progress in Fourier transform, 19.5288% done!
Making progress in Fourier transform, 19.834% done!
Making progress in Fourier transform, 20.1392% done!
Making progress in Fourier transform, 20.4443% done!
Making progress in Fourier transform, 20.7495% done!
Making progress in Fourier transform, 21.0547% done!
Making progress in Fourier transform, 21.3599% done!
Making progress in Fourier transform, 21.6651% done!
Making progress in Fourier transform, 21.9703% done!
Making progress in Fourier transform, 22.2755% done!
```

```
Making progress in Fourier transform, 22.5806% done!
Making progress in Fourier transform, 22.8858% done!
Making progress in Fourier transform, 23.191% done!
Making progress in Fourier transform, 23.4962% done!
Making progress in Fourier transform, 23.8014% done!
Making progress in Fourier transform, 24.1066% done!
Making progress in Fourier transform, 24.4118% done!
Making progress in Fourier transform, 24.7169% done!
Making progress in Fourier transform, 25.0221% done!
Making progress in Fourier transform, 25.3273% done!
Making progress in Fourier transform, 25.6325% done!
Making progress in Fourier transform, 25.9377% done!
Making progress in Fourier transform, 26.2429% done!
Making progress in Fourier transform, 26.5481% done!
Making progress in Fourier transform, 26.8532% done!
Making progress in Fourier transform, 27.1584% done!
Making progress in Fourier transform, 27.4636% done!
Making progress in Fourier transform, 27.7688% done!
Making progress in Fourier transform, 28.074% done!
Making progress in Fourier transform, 28.3792% done!
Making progress in Fourier transform, 28.6843% done!
Making progress in Fourier transform, 28.9895% done!
Making progress in Fourier transform, 29.2947% done!
Making progress in Fourier transform, 29.5999% done!
Making progress in Fourier transform, 29.9051% done!
Making progress in Fourier transform, 30.2103% done!
Making progress in Fourier transform, 30.5155% done!
Making progress in Fourier transform, 30.8206% done!
Making progress in Fourier transform, 31.1258% done!
Making progress in Fourier transform, 31.431% done!
Making progress in Fourier transform, 31.7362% done!
Making progress in Fourier transform, 32.0414% done!
Making progress in Fourier transform, 32.3466% done!
Making progress in Fourier transform, 32.6518% done!
Making progress in Fourier transform, 32.9569% done!
Making progress in Fourier transform, 33.2621% done!
Making progress in Fourier transform, 33.5673% done!
Making progress in Fourier transform, 33.8725% done!
Making progress in Fourier transform, 34.1777% done!
Making progress in Fourier transform, 34.4829% done!
Making progress in Fourier transform, 34.788% done!
Making progress in Fourier transform, 35.0932% done!
Making progress in Fourier transform, 35.3984% done!
Making progress in Fourier transform, 35.7036% done!
Making progress in Fourier transform, 36.0088% done!
Making progress in Fourier transform, 36.314% done!
```

```
Making progress in Fourier transform, 36.6192% done!
Making progress in Fourier transform, 36.9243% done!
Making progress in Fourier transform, 37.2295% done!
Making progress in Fourier transform, 37.5347% done!
Making progress in Fourier transform, 37.8399% done!
Making progress in Fourier transform, 38.1451% done!
Making progress in Fourier transform, 38.4503% done!
Making progress in Fourier transform, 38.7555% done!
Making progress in Fourier transform, 39.0606% done!
Making progress in Fourier transform, 39.3658% done!
Making progress in Fourier transform, 39.671% done!
Making progress in Fourier transform, 39.9762% done!
Making progress in Fourier transform, 40.2814% done!
Making progress in Fourier transform, 40.5866% done!
Making progress in Fourier transform, 40.8918% done!
Making progress in Fourier transform, 41.1969% done!
Making progress in Fourier transform, 41.5021% done!
Making progress in Fourier transform, 41.8073% done!
Making progress in Fourier transform, 42.1125% done!
Making progress in Fourier transform, 42.4177% done!
Making progress in Fourier transform, 42.7229% done!
Making progress in Fourier transform, 43.028% done!
Making progress in Fourier transform, 43.3332% done!
Making progress in Fourier transform, 43.6384% done!
Making progress in Fourier transform, 43.9436% done!
Making progress in Fourier transform, 44.2488% done!
Making progress in Fourier transform, 44.554% done!
Making progress in Fourier transform, 44.8592% done!
Making progress in Fourier transform, 45.1643% done!
Making progress in Fourier transform, 45.4695% done!
Making progress in Fourier transform, 45.7747% done!
Making progress in Fourier transform, 46.0799% done!
Making progress in Fourier transform, 46.3851% done!
Making progress in Fourier transform, 46.6903% done!
Making progress in Fourier transform, 46.9955% done!
Making progress in Fourier transform, 47.3006% done!
Making progress in Fourier transform, 47.6058% done!
Making progress in Fourier transform, 47.911% done!
Making progress in Fourier transform, 48.2162% done!
Making progress in Fourier transform, 48.5214% done!
Making progress in Fourier transform, 48.8266% done!
Making progress in Fourier transform, 49.1317% done!
Making progress in Fourier transform, 49.4369% done!
Making progress in Fourier transform, 49.7421% done!
Making progress in Fourier transform, 50.0473% done!
Making progress in Fourier transform, 50.3525% done!
```

```
Making progress in Fourier transform, 50.6577% done!
Making progress in Fourier transform, 50.9629% done!
Making progress in Fourier transform, 51.268% done!
Making progress in Fourier transform, 51.5732% done!
Making progress in Fourier transform, 51.8784% done!
Making progress in Fourier transform, 52.1836% done!
Making progress in Fourier transform, 52.4888% done!
Making progress in Fourier transform, 52.794% done!
Making progress in Fourier transform, 53.0992% done!
Making progress in Fourier transform, 53.4043% done!
Making progress in Fourier transform, 53.7095% done!
Making progress in Fourier transform, 54.0147% done!
Making progress in Fourier transform, 54.3199% done!
Making progress in Fourier transform, 54.6251% done!
Making progress in Fourier transform, 54.9303% done!
Making progress in Fourier transform, 55.2355% done!
Making progress in Fourier transform, 55.5406% done!
Making progress in Fourier transform, 55.8458% done!
Making progress in Fourier transform, 56.151% done!
Making progress in Fourier transform, 56.4562% done!
Making progress in Fourier transform, 56.7614% done!
Making progress in Fourier transform, 57.0666% done!
Making progress in Fourier transform, 57.3717% done!
Making progress in Fourier transform, 57.6769% done!
Making progress in Fourier transform, 57.9821% done!
Making progress in Fourier transform, 58.2873% done!
Making progress in Fourier transform, 58.5925% done!
Making progress in Fourier transform, 58.8977% done!
Making progress in Fourier transform, 59.2029% done!
Making progress in Fourier transform, 59.508% done!
Making progress in Fourier transform, 59.8132% done!
Making progress in Fourier transform, 60.1184% done!
Making progress in Fourier transform, 60.4236% done!
Making progress in Fourier transform, 60.7288% done!
Making progress in Fourier transform, 61.034% done!
Making progress in Fourier transform, 61.3392% done!
Making progress in Fourier transform, 61.6443% done!
Making progress in Fourier transform, 61.9495% done!
Making progress in Fourier transform, 62.2547% done!
Making progress in Fourier transform, 62.5599% done!
Making progress in Fourier transform, 62.8651% done!
Making progress in Fourier transform, 63.1703% done!
Making progress in Fourier transform, 63.4754% done!
Making progress in Fourier transform, 63.7806% done!
Making progress in Fourier transform, 64.0858% done!
Making progress in Fourier transform, 64.391% done!
```

```
Making progress in Fourier transform, 64.6962% done!
Making progress in Fourier transform, 65.0014% done!
Making progress in Fourier transform, 65.3066% done!
Making progress in Fourier transform, 65.6117% done!
Making progress in Fourier transform, 65.9169% done!
Making progress in Fourier transform, 66.2221% done!
Making progress in Fourier transform, 66.5273% done!
Making progress in Fourier transform, 66.8325% done!
Making progress in Fourier transform, 67.1377% done!
Making progress in Fourier transform, 67.4429% done!
Making progress in Fourier transform, 67.748% done!
Making progress in Fourier transform, 68.0532% done!
Making progress in Fourier transform, 68.3584% done!
Making progress in Fourier transform, 68.6636% done!
Making progress in Fourier transform, 68.9688% done!
Making progress in Fourier transform, 69.274% done!
Making progress in Fourier transform, 69.5791% done!
Making progress in Fourier transform, 69.8843% done!
Making progress in Fourier transform, 70.1895% done!
Making progress in Fourier transform, 70.4947% done!
Making progress in Fourier transform, 70.7999% done!
Making progress in Fourier transform, 71.1051% done!
Making progress in Fourier transform, 71.4103% done!
Making progress in Fourier transform, 71.7154% done!
Making progress in Fourier transform, 72.0206% done!
Making progress in Fourier transform, 72.3258% done!
Making progress in Fourier transform, 72.631% done!
Making progress in Fourier transform, 72.9362% done!
Making progress in Fourier transform, 73.2414% done!
Making progress in Fourier transform, 73.5466% done!
Making progress in Fourier transform, 73.8517% done!
Making progress in Fourier transform, 74.1569% done!
Making progress in Fourier transform, 74.4621% done!
Making progress in Fourier transform, 74.7673% done!
Making progress in Fourier transform, 75.0725% done!
Making progress in Fourier transform, 75.3777% done!
Making progress in Fourier transform, 75.6829% done!
Making progress in Fourier transform, 75.988% done!
Making progress in Fourier transform, 76.2932% done!
Making progress in Fourier transform, 76.5984% done!
Making progress in Fourier transform, 76.9036% done!
Making progress in Fourier transform, 77.2088% done!
Making progress in Fourier transform, 77.514% done!
Making progress in Fourier transform, 77.8191% done!
Making progress in Fourier transform, 78.1243% done!
Making progress in Fourier transform, 78.4295% done!
```

```
Making progress in Fourier transform, 78.7347% done!
Making progress in Fourier transform, 79.0399% done!
Making progress in Fourier transform, 79.3451% done!
Making progress in Fourier transform, 79.6503% done!
Making progress in Fourier transform, 79.9554% done!
Making progress in Fourier transform, 80.2606% done!
Making progress in Fourier transform, 80.5658% done!
Making progress in Fourier transform, 80.871% done!
Making progress in Fourier transform, 81.1762% done!
Making progress in Fourier transform, 81.4814% done!
Making progress in Fourier transform, 81.7866% done!
Making progress in Fourier transform, 82.0917% done!
Making progress in Fourier transform, 82.3969% done!
Making progress in Fourier transform, 82.7021% done!
Making progress in Fourier transform, 83.0073% done!
Making progress in Fourier transform, 83.3125% done!
Making progress in Fourier transform, 83.6177% done!
Making progress in Fourier transform, 83.9228% done!
Making progress in Fourier transform, 84.228% done!
Making progress in Fourier transform, 84.5332% done!
Making progress in Fourier transform, 84.8384% done!
Making progress in Fourier transform, 85.1436% done!
Making progress in Fourier transform, 85.4488% done!
Making progress in Fourier transform, 85.754% done!
Making progress in Fourier transform, 86.0591% done!
Making progress in Fourier transform, 86.3643% done!
Making progress in Fourier transform, 86.6695% done!
Making progress in Fourier transform, 86.9747% done!
Making progress in Fourier transform, 87.2799% done!
Making progress in Fourier transform, 87.5851% done!
Making progress in Fourier transform, 87.8903% done!
Making progress in Fourier transform, 88.1954% done!
Making progress in Fourier transform, 88.5006% done!
Making progress in Fourier transform, 88.8058% done!
Making progress in Fourier transform, 89.111% done!
Making progress in Fourier transform, 89.4162% done!
Making progress in Fourier transform, 89.7214% done!
Making progress in Fourier transform, 90.0266% done!
Making progress in Fourier transform, 90.3317% done!
Making progress in Fourier transform, 90.6369% done!
Making progress in Fourier transform, 90.9421% done!
Making progress in Fourier transform, 91.2473% done!
Making progress in Fourier transform, 91.5525% done!
Making progress in Fourier transform, 91.8577% done!
Making progress in Fourier transform, 92.1628% done!
Making progress in Fourier transform, 92.468% done!
```

```
Making progress in Fourier transform, 92.7732% done!
Making progress in Fourier transform, 93.0784% done!
Making progress in Fourier transform, 93.3836% done!
Making progress in Fourier transform, 93.6888% done!
Making progress in Fourier transform, 93.994% done!
Making progress in Fourier transform, 94.2991% done!
Making progress in Fourier transform, 94.6043% done!
Making progress in Fourier transform, 94.9095% done!
Making progress in Fourier transform, 95.2147% done!
Making progress in Fourier transform, 95.5199% done!
Making progress in Fourier transform, 95.8251% done!
Making progress in Fourier transform, 96.1303% done!
Making progress in Fourier transform, 96.4354% done!
Making progress in Fourier transform, 96.7406% done!
Making progress in Fourier transform, 97.0458% done!
Making progress in Fourier transform, 97.351% done!
Making progress in Fourier transform, 97.6562% done!
Making progress in Fourier transform, 97.9614% done!
Making progress in Fourier transform, 98.2665% done!
Making progress in Fourier transform, 98.5717% done!
Making progress in Fourier transform, 98.8769% done!
Making progress in Fourier transform, 99.1821% done!
Making progress in Fourier transform, 99.4873% done!
Making progress in Fourier transform, 99.7925% done!
measured 26527, approximation for 0.40477 is 1205 / 2977
Odd denominator, expanding by 2
possible period is 5954
83^2977 + 1 mod 129 = 24,
83^2977 - 1 mod 129 = 22
129 = 3 * 43
```